

VISIT...

LANZAROTE
Caliente.COM

Database Programming with C#

CARSTEN THOMSEN

Apress™

Database Programming with C#

Copyright © 2002 by Carsten Thomsen

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-010-4

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewer: Douglas Milnes

Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore,
Karen Watterson, John Zukowski

Managing Editor: Grace Wong

Copy Editors: Nicole LeClerc, Ami Knox

Production Editor: Tory McLearn

Compositor: Impressions Book and Journal Services, Inc.

Indexer: Valerie Haynes Perry

Cover Designer: Tom Debolski

Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit <http://www.springer-ny.com>.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit <http://www.springer.de>.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710.

Email info@apress.com or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section.

You will need to answer questions pertaining to this book in order to successfully download the code.

Using Stored Procedures, Views, and Triggers

How to Use Stored Procedures, Views, and Triggers

SERVER-SIDE PROCESSING, which is when you let a server process your queries and the like, is probably a concept you have heard of and it's the very topic of this chapter. Well, to some extent anyway. I discuss three specific ways of doing server-side processing: stored procedures, triggers, and views. The good thing about server-side processing is that you can use the power and resources of your server for doing purely data-related processing and thus leave your client free to do other stuff, and your network clearer of data that the client doesn't want. It's not always appropriate to do so, but in many cases you can benefit from it.

This chapter includes several hands-on exercises that will take you through creating stored procedures, views, and triggers. See the Exercise items that appear throughout the text.

Although this chapter primarily focuses on SQL Server 2000 features, some of the functionality can certainly be reproduced in the other DBMSs I cover in this book:

- *SQL Server 7.0:* All functionality shown in this chapter can be reproduced. However, SQL Server 7.0 doesn't support the INSTEAD OF triggers described in "Using Triggers."
- *Microsoft Access:* Microsoft Access doesn't support stored procedures or triggers. However, views can be reproduced as queries in Microsoft Access, but you can't do this from within the VS .NET IDE; you have to use other means, like the Microsoft Access front-end. If you are unfamiliar with

Microsoft Access, I can recommend you read the following book to get you up to speed: *From Access to SQL Server*, by Russell Sinclair. Published by Apress, September 2000. ISBN: 1893115-240.

- *Oracle*: Oracle supports all the server-side processing described in this chapter.
- *MySQL*: For the examples in this book, I have been using MySQL version 3.23.45, which doesn't support triggers, views, or stored procedures, meaning there is no example code for MySQL in this chapter. However, at the time of writing (March 2002), an alpha version (4.0) of MySQL is available for download from <http://www.mysql.com>. The final version 4.0 is supposed to support stored procedures, views, and triggers. Even when these server-side processing means are available in MySQL, it's still not possible to create any of these items from within the VS .NET IDE.

The code for this chapter has examples for all the listed DBMSs where appropriate.

Optimization Issues

When I talk about optimizing performance of an application, there are a number of things to consider, but let's just make one thing clear before I go on: I am only talking distributed applications and not stand-alone applications that sit nicely on a possibly disconnected single PC. These stand-alone applications are also called *single tier* or *monolithic applications*.¹ The applications I discuss here use a network of some sort to access data and business services.

Okay, now that the basics are out of the way, I can focus on the obstacles that can lead to decreasing performance and how you need to know these obstacles well when you start the optimization process. You should keep such obstacles in mind when you design your application. However, the various resources, such as network bandwidth, processor power, available RAM, and so on, most often change over time, and then you'll have to reconsider if your application needs changing.

Table 6-1 lists all the varying factors that can influence the performance of your application, which could be a topic for an entire book. However, although I only describe these factors briefly, I want you to be aware of the resources mentioned; they have great influence on what server-side processing resources you should choose when you design your application. In general, it's often the client queries and not the server itself that create the biggest performance problems.

1. Stand-alone applications don't have to be single tier, but they generally are.

Table 6-1. Performance Resources Optimization

RESOURCE NAME	DESCRIPTION
Network resources	When speaking of network resources, I am referring to the actual bandwidth of the network. Consider your network setup—whether you are on a LAN or you are accessing resources over a WAN such as the Internet, and so on. If you have a low bandwidth, it's obvious that you want to transfer as little data across the network as possible. If on the other hand you have plenty of bandwidth, you might want to transfer large amounts of data across the network. However, best practices prescribe that you only transfer the data needed across your network, even when you have wide bandwidth.
Local processing resources	If you have the raw power available on your local box, it can be good to do most of the data processing there. Mind you, it all depends on the available bandwidth and the processing resources on the server.
Server processing resources	Server-side processing is desirable, if the server has resources to do so. Another thing you should consider is whether it has the resources to serve all your clients, if you let the server do some of the data processing.
Data distribution	Although strictly speaking this isn't a resource as such, it's definitely another issue you might need to consider. If your data comes from various different and even disparate data sources, it often doesn't make too much sense to have one server process data from all the data sources, just to send the result set to the client. In most cases, it makes sense to have all the data delivered directly to the client.

Table 6-1 just provides a quick overview. Table 6-2 shows you some different application scenarios.

Table 6-2. Different Application Scenarios

CLIENT MACHINE	SERVER	NETWORK	RECOMMENDATION
Limited processing resources	Plenty of processing resources	Limited bandwidth	Now, this one is obvious. You should use the raw processing power of the server to process the data and only return the requested data. This will save resources on the network and on the client.
Plenty of processing resources	Plenty of processing resources	Limited bandwidth	Hmm, processing could be done on either the client or the server, but it really depends on the amount of data you need to move across the network. If it's a limited amount of data, processing on either side will do, but if it's a lot of data, then let the server do the processing. Another solution could be to store the data locally and then use replication or batch processing to update the server.
Plenty of processing resources	Limited processing resources	Limited bandwidth	In this case, processing should be done on the client, but it really depends on the amount of data you need to move across the network. If it's a limited amount of data, the client should do the processing; but if it's a lot of data, you might consider letting the server do some of the processing, or even better; upgrade your server.
Plenty of processing resources	Limited processing resources	Plenty of bandwidth	Okay, don't think too hard about this one—processing should be done on the client.

I could add plenty more scenarios to the list, but I think you get the picture. You'll rarely encounter a situation that matches a straightforward scenario with a simple answer. It's your job to know about all the potential issues when you design your application and have to decide on where to process your data. Quite often different aspects of an application have different data processing needs, so the answer may vary even within a single application. One book that will help

you with many common problems you may encounter with SQL Server is this one:

- *SQL Server: Common Problems, Tested Solutions*, by Neil Pike. Published by Apress, October 2000. ISBN: 189311581X.

Troubleshooting Performance Degradation

When you realize that you have performance problems or when you just want to optimize your server, you need one or more tools to help. SQL Server and Windows NT/2000 provides a number of tools you can use when troubleshooting and here are a few of them:

- Database Consistency Checker (DBCC) (SQL Server)
- Performance Monitor (Windows NT/2000)
- Query Analyzer (SQL Server)
- System Stored Procedures (SQL Server)

I'll briefly describe what you can use these tools for and give you links for obtaining more information.

Database Consistency Checker

The *Database Consistency Checker* (DBCC) is used for checking the logic as well as the consistency of your databases using T-SQL DBCC statements. Furthermore, many of the DBCC statements can also fix the problems detected when running. DBCC statements are T-SQL enhancements and as such must be run as SQL scripts. Here is one example of a DBCC statement:

```
DBCC CHECKDB
```

This DBCC statement is used for checking the structural integrity of the objects in the database you specify. It can also fix the problems found when running. There are many DBCC statements, and this isn't the place to go over these, but check SQL Server Books Online (included with SQL Server) for more information about DBCC.

Performance Monitor

The *Performance Monitor* (perfmon) is used for tracking and recording activity on your machine or rather any machine within your enterprise. perfmon comes with Windows NT/2000/XP and is located in the Administrative Tools menu, but you can also run it from a command prompt, or the Run facility of Windows Start Menu, by executing perfmon. Any of the Windows platforms mentioned produces counters that can be tracked or polled by perfmon at regular intervals if needed. SQL Server also comes with counters that can be tracked or polled by perfmon. Some of the more general counters are used for polling processor time, disk access, memory usage, and so on. Arguably the best of it all is the ability to save a session of all activity recorded or polled within any given time frame. You can then play back a saved session, whenever appropriate. This is especially important when you want to establish a baseline against which to compare future session recordings.

Check your Windows NT/2000/XP documentation for more information about perfmon.

Query Analyzer

The *Query Analyzer* is an external tool that comes with SQL Server for analyzing and optimizing your queries. You can find it in the menus created by SQL Server Setup.

Query Analyzer can be used for validating your queries in the form of script files and queries you type yourself in the query window. Besides validating a query, you can get Query Analyzer to analyze it by running. The analysis includes an execution plan, statistics, and a trace of the query being executed. Queries can get complicated, and many do when joining tables, and it isn't always obvious how much processing a particular query will take. There's normally more than one way to get to complex data, so the trace is invaluable in optimizing your data requests.

See SQL Server Books Online (included with SQL Server) for more information about Query Analyzer. You can actually invoke the Query Analyzer part of the SQL Server Books Online help text from within Query Analyzer by pressing F1.

System Stored Procedures

The *System Stored Procedures* is a set of stored procedures that comes with SQL Server for database administrators to use for maintaining and administering SQL Server. There are a number of System Stored Procedures, including two XML ones, and I certainly can't cover them here, but I can mention some of the

functionality they cover: they let you see who's logged on to the system, administer registration with Active Directory, set up replication, set up full-text search, create and edit maintenance plans, and administer a database in general.

See SQL Server Books Online (comes with SQL Server) for more information about the System Stored Procedures.

Using Stored Procedures

A *stored procedure* is a precompiled batch² of SQL statement(s) that is stored on the database server. The SQL statements are always executed on the database server. Stored procedures have long been a good way of letting the server process your data. They can significantly reduce the workload on the client, and once you get to know them you'll wonder how you ever managed without them.

There is certainly more to a stored procedure than just mentioned, but I do think this is the most significant aspect of a stored procedure. Think about it: it's a way of grouping a batch of SQL statements, storing it on the database server, and executing it with a single call. The fact that the stored procedure is precompiled will save you time as well when executed. Furthermore, the stored procedure can be executed by any number of users, meaning you might save a lot of bandwidth just by calling the stored procedure instead of sending the whole SQL statement every time.

A stored procedure can contain any SQL statement that your database server can understand. This means you can use stored procedures for various tasks, such as executing queries—both so-called action queries, such as DELETE queries, and row-returning queries, such as SELECT statements.

Another task you can use a stored procedure for is database maintenance. Use it to run cleanup SQL statements when the server is least busy and thus save the time and effort of having to do this manually. I won't cover maintenance tasks in this chapter, but they are important, and you should be aware of the various tasks you can perform with stored procedures. If you're like me, you have been or are working for a small company that doesn't have a database administrator, in which case you're in charge of keeping the database server running. Granted, it's not an ideal situation, but you certainly get to know your DBMS in different ways than you would just being a programmer, and that's not bad at all.

To sum it up: a stored procedure is a precompiled SQL statement or batch of SQL statements that is stored on the database server. All processing takes place on the server, and any result requested by a client is then returned in a prepared format.

2. Actually some stored procedures only hold one SQL statement.

Why Use a Stored Procedure?

You should use a stored procedure in the following cases (please note that other cases do apply, depending on your circumstances):

- Executing one or more related SQL statements on a regular basis
- Hiding complex table structures from client developers
- Returning the result of your SQL statements because you have a limited bandwidth on your network
- Delegating data processing to the server because you have limited processing resources on your client
- Ensuring processes are run, on a scheduled basis, without user intervention

Granted, there can be substantially more work in setting up a stored procedure than in just executing the SQL statement(s) straight from the client, but my experience has confirmed that the extra work saves you at least tenfold the time once you start coding and using your application. Even SQL Server itself and other major DBMSs use stored procedures for maintenance and other administrative tasks.

One last thing I want to mention is the fact that if you base a lot of your data calls on stored procedures, it can be much easier to change the data calls at a later date. You can simply change the stored procedure and not the application itself, meaning you don't have to recompile a business service or even your client application, depending on how you have designed your application. On the negative side, stored procedures are often written using database vendor-specific SQL extensions, which mean that they're hard to migrate to a different RDBMS. This of course is only a real concern if you're planning to move to another RDBMS.

Planning a Move to a Different RDBMS

If you're planning to move to another RDBMS from SQL Server, or just want to make it as easy as possible should management decide so in the future, it'll probably be a good idea to look up the following T-SQL statements in the SQL Server Books Online Help Documentation:

- **SET ANSI_DEFAULTS:** This statement sets the ANSI defaults on for the duration of the query session, trigger, or stored procedure.
- **SET FIPS_FLAGGER:** This statement can be used to check for compliance with the ANSI SQL-92 standard.

If you use these statements appropriately, they can certainly help ease the move from SQL Server to another ANSI SQL-92-compliant RDBMS.

Creating and Running a Stored Procedure

Creating a stored procedure is fairly easy, and you're probably used to working with the Enterprise Manager that comes with SQL Server or a different stored procedure editor for SQL Server or Oracle. If this is the case, you may want to check out the facilities in the Server Explorer in the VS .NET IDE. Among other things, it's much easier to run and test a stored procedure directly from the text editor. Anyway, here's how you would create a stored procedure for the example UserMan database:

1. Open up the Server Explorer window.
2. Expand the UserMan database on your database server.
3. Right-click the Stored Procedures node and select New Stored Procedure.

This brings up the Stored Procedure text editor, which incidentally looks a lot like your C# code editor. Except for syntax checking and other minor stuff, they are exactly the same (see Figure 6-1).

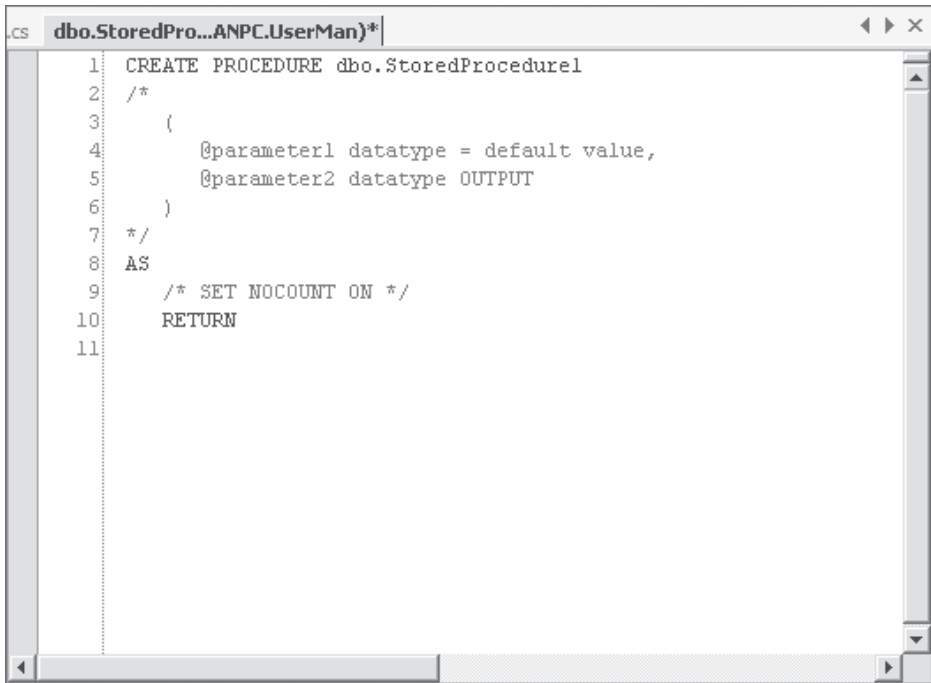


Figure 6-1. Stored procedure editor with SQL Server default template

NOTE With SQL Server it's only possible to use T-SQL for your stored procedures. However, the upcoming version of SQL Server, code-named Yukon, will have support for the .NET programming languages. Knowing this, perhaps you'll want to create your stored procedures in C# or VB .NET.

Creating a Simple Stored Procedure

Once you've created a stored procedure, you need to give it a name. As you can see from your stored procedure editor, the template automatically names it `StoredProcedure1`. If you're wondering about the `dbo` prefix, it simply means that the stored procedure is created for the `dbo` user. In SQL Server terms, `dbo` stands for *database owner*, and it indicates who owns the database object, which is a stored procedure in this case. If you've been working with SQL Server for a while, you probably know the term *broken ownership chain*. An ownership chain is the dependency of a stored procedure upon tables, views, or other stored procedures.

Generally, the objects that a view or stored procedure depend on are also owned by the owner of the view or stored procedure. In such a case there are no problems, because SQL Server doesn't check permissions in this situation. (It doesn't really have to, does it?) However, when one or more of the dependent database objects are owned by a different user than the one owning the view or stored procedure, the ownership chain is said to be broken. This means that SQL Server has to check the permissions of any dependent database object that has a different owner. This can be avoided, if the same user, such as `dbo`, owns all of your database objects. I am not telling you to do it this way, but it's one option available to you.

Okay, let's say you've deleted the `StoredProcedure1` name and replaced it with `SimpleStoredProcedure`. To save the stored procedure before continuing, press `Ctrl+S`. If you saved your stored procedure at this point, you would notice that you don't have to name it using a `Save As` dialog box, because you've already named it. The editor will make sure that the stored procedure is saved on the database server with the name you've entered, which in this case is `SimpleStoredProcedure`. You shouldn't save it until you've renamed it, because you'll end up having to remove unwanted stored procedures.

Although you can see your stored procedure in the `Stored Procedure` folder of the `SQL Server Enterprise Manager` and the `Stored Procedure` node in the `Server Explorer`, there isn't actually an area in your database designated for just stored procedures. The stored procedure is saved to the system tables as are most other objects in SQL Server.

As soon as you have saved it, the very first line of the stored procedure changes. The SQL statement `CREATE PROCEDURE` is changed so that the first line reads:

```
ALTER PROCEDURE dbo.SimpleStoredProcedure
```

Why? Well, you just saved the newly created stored procedure, which means that you can't create another with the same name. Changing `CREATE` to `ALTER` takes care of that. It's that simple. In case you're wondering what happens when you change the name of your stored procedure and the SQL statement still reads `ALTER PROCEDURE . . .`, I can tell you: the editor takes care of it for you and creates a new procedure. Try it and see for yourself! Basically, this means that `CREATE PROCEDURE` is never actually needed; one can simply use `ALTER PROCEDURE`, even on brand new procedures. However, this can be a dangerous practice, if you inadvertently change the name of your stored procedure to the name of an already existing one.

The `SimpleStoredProcedure` doesn't actually do a lot, does it? Okay, let me show you how to change that. In Figure 6-1, you can see two parts of the stored procedure: The first part is the header and then there is the actual stored procedure itself. The header consists of all text down to and including Line 7. Basically,

the header declares how the stored procedure should be called, how many arguments to include and what type of arguments, and so on. Since this is a very simple procedure, I don't want any arguments, so I'll leave the commented-out text alone.

If you haven't changed the default editor settings, text that is commented out or any comments you have inserted yourself are printed in green. In a SQL Server stored procedure, comments are marked using start and end tags: `/*` for the comment start tag and `*/` for the comment end tag. This has one advantage over the way you insert comments in your C# code in that you don't have to have a comment start tag on every line you want to comment out. You only need to have both a start and end tag.

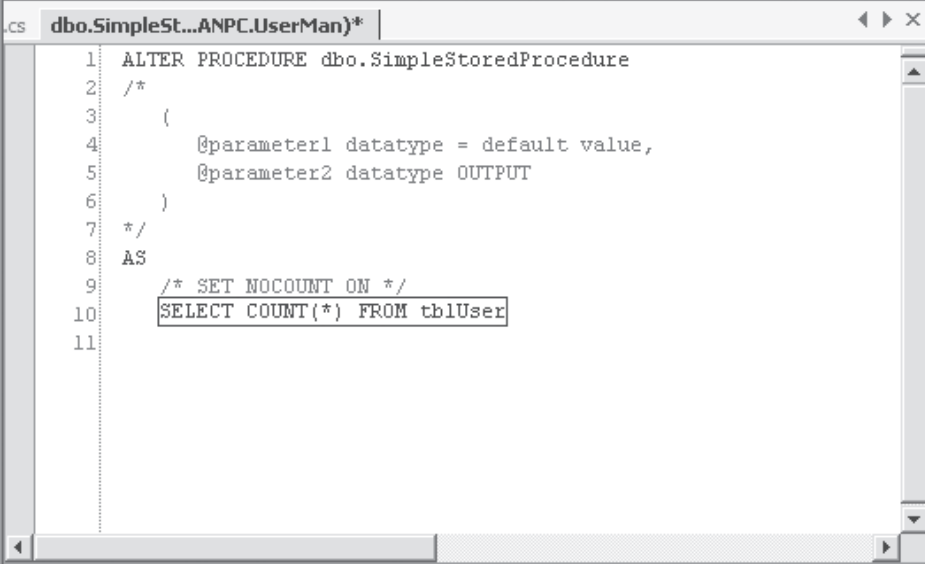
The second part of the stored procedure is the part that starts with the `AS` clause on Line 8. The `AS` clause indicates that the text that follows is the body of the stored procedure, the instructions on what to do when the stored procedure is called and executed.

EXERCISE

- 1) Create a stored procedure, name it **SimpleStoredProcedure**, and save it as described earlier.
- 2) Type the following text on Line 10 in the `SimpleStoredProcedure` in place of the `RETURN` statement:

```
SELECT COUNT(*) FROM tblUser
```

Now the stored procedure should look like the example in Figure 6-2. The stored procedure will return the number of rows in the `tblUser` table. Please note that it's generally good practice to keep the `RETURN` statement as part of your stored procedure, but I'm taking it out and leaving it for an explanation later, when I discuss return values and how they're handled in code.



```
1 ALTER PROCEDURE dbo.SimpleStoredProcedure
2 /*
3 (
4     @parameter1 datatype = default value,
5     @parameter2 datatype OUTPUT
6 )
7 */
8 AS
9     /* SET NOCOUNT ON */
10    SELECT COUNT(*) FROM tblUser
11
```

Figure 6-2. Stored procedure that returns the number of rows in the `tblUser` table

3) Don't forget to save the changes using Ctrl+S.

Running a Simple Stored Procedure from the IDE

Of course, there's no point in having a stored procedure that just sits there, so here's what you do to run it: if you have the stored procedure open in the stored procedure editor window, you can right-click anywhere in the editor window and select Run Stored Procedure from the pop-up menu. If you do this with the stored procedure you created in the exercise in the previous section, the Output window, located just below the editor window, should display the output from the stored procedure as shown in Figure 6-3.

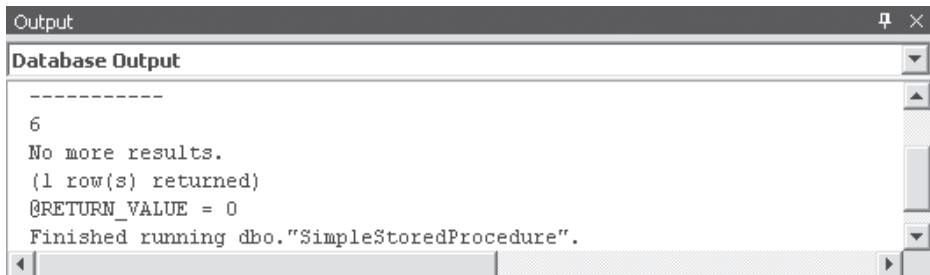


Figure 6-3. The Output window with output from SimpleStoredProcedure

If you have closed down the stored procedure editor window, you can run the stored procedure from the Server Explorer. Expand the database node, right-click the Stored Procedures node, and select Run Stored Procedure from the pop-up menu. This will execute the stored procedure the exact same way as if you were running it from the editor window.

Running a Simple Stored Procedure from Code

Okay, now that you have a fully functional stored procedure, you can try and run it from code. Listing 6-1 shows you some very simple code that will run the stored procedure. The example code in this listing uses data classes that were introduced in Chapters 3A and 3B.

Listing 6-1. Running a Simple Stored Procedure

```

1 public void ExecuteSimpleSP() {
2     SqlConnection cnnUserMan;
3     SqlCommand cmmUser;
4     object objNumUsers;
5
6     // Instantiate and open the connection
7     cnnUserMan = new SqlConnection(STR_CONNECTION_STRING);
8     cnnUserMan.Open();
9
10    // Instantiate and initialize command
11    cmmUser = new SqlCommand("SimpleStoredProcedure", cnnUserMan);
12    cmmUser.CommandType = CommandType.StoredProcedure;
13
14    objNumUsers = cmmUser.ExecuteScalar();
15    MessageBox.Show(objNumUsers.ToString());
16 }
```

The code in Listing 6-1 retrieves the return value from the stored procedure. Now, this isn't usually all you want from a stored procedure, but it merely demonstrates what a simple stored procedure looks like. The stored procedure itself could just as well have had a `DELETE FROM tblUser WHERE LastName='Johnson'` SQL statement. If you want to execute this from code, you need to know if the stored procedure returns a value or not. It doesn't in this case, so you need to use the **ExecuteNonQuery** method of the **SqlCommand** class.

EXERCISE

- 1) Create a new stored procedure and save it with the name **uspGetUsers**.
- 2) Type in the following text on Line 10 in place of the RETURN statement:

SELECT * FROM tblUser

Now the stored procedure should look like the one in Figure 6-4. This stored procedure will return all rows in the `tblUser` table.

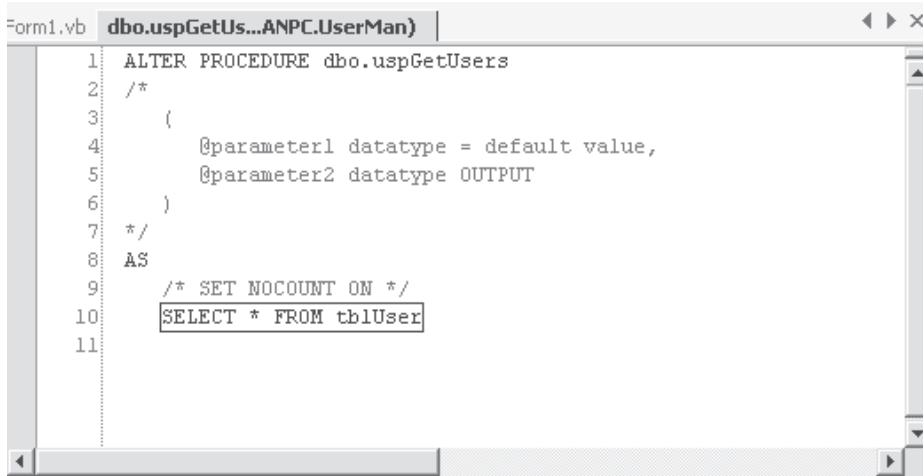


Figure 6-4. The `uspGetUsers` stored procedure

- 3) Don't forget to save the changes using **Ctrl+S**.

What you need now is some code to retrieve the rows from the stored procedure (see Listing 6-2).

Listing 6-2. Retrieving Rows from a Stored Procedure

```

1 public void ExecuteSimpleRowReturningSP() {
2     SqlConnection cnnUserMan;
3     SqlCommand cmmUser;
4     SqlDataReader drdUser;
5
6     // Instantiate and open the connection
7     cnnUserMan = new SqlConnection(STR_CONNECTION_STRING);
8     cnnUserMan.Open();
9
10    // Instantiate and initialize command
11    cmmUser = new SqlCommand("uspGetUsers", cnnUserMan);
12    cmmUser.CommandType = CommandType.StoredProcedure;
13
14    // Retrieve all user rows
15    drdUser = cmmUser.ExecuteReader();
16 }
```

The example in Listing 6-2 retrieves the rows returned from the stored procedure by using the **ExecuteReader** method of the **SqlCommand** class. Please note that this method and the related **ExecuteXmlReader** method are the only options for retrieving rows as the result of a function call with the Command class.

Creating a Stored Procedure with Arguments

Sometimes it's a good idea to create a stored procedure with arguments³ instead of having more stored procedures essentially doing the same. It also gives you some flexibility with regards to making minor changes to your application without having to recompile one or more parts of it, because you can add to the number of arguments and keep existing applications running smoothly by specifying a default value for the new arguments.

Another reason for using arguments with stored procedures is to make the stored procedure behave differently, depending on the input from the arguments. One argument might hold the name of a table, view, or another stored procedure to extract data from.

3. I'm using the word *argument* here, but I might as well call it *parameter*, like T-SQL does. However, the two words are synonymous in this case.

TIP In SQL Server you can use the `EXECUTE sp_executesql` statement and System Stored Procedure with arguments of type ***ntext***, ***nchar***, or ***nvarchar*** to execute parameterized queries. See the SQL Server Books Online Help Documentation for more information.

EXERCISE

- 1) Create a new stored procedure and save it with the name **uspGetUsersByLastName**.
- 2) Type in the following text on Lines 10 and 11 in place of the RETURN statement:

```
SELECT * FROM tblUser  
WHERE LastName = @strLastName
```

- 3) Uncomment Lines 2 to 7, and insert the following text instead of Lines 3 and 4:

```
@strLastName varchar(50)
```

The stored procedure should look like the one in Figure 6-5. This stored procedure will return all rows in the `tblUser` table where the `LastName` column matches the `strLastName` argument.

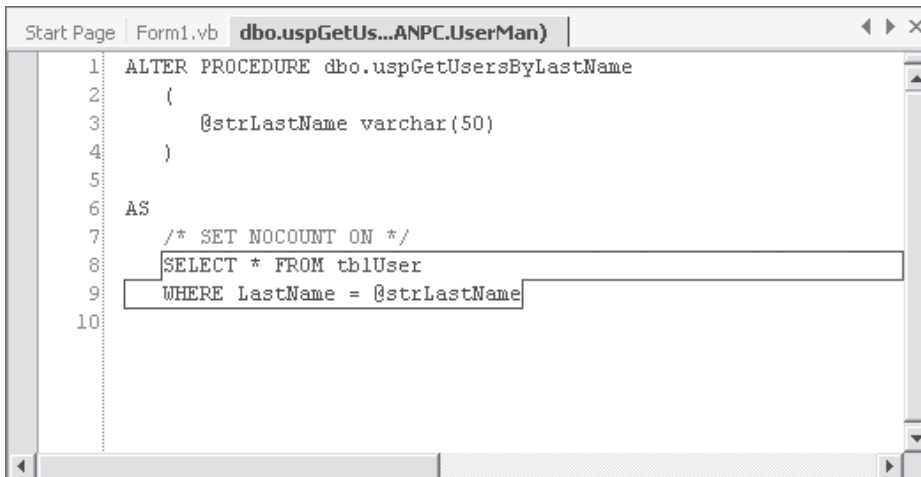


Figure 6-5. The `uspGetUsersByLastName` stored procedure

- 4) Don't forget to save your changes using **Ctrl+S**.

Arguments in stored procedures can be either input or output. If you include an input argument, you don't have to specify anything after the data type, but if you use an output argument, you need to specify the OUTPUT keyword after the data type.

NOTE *I only cover the absolute basics of how to create a stored procedure in this chapter. If you need more information, I suggest you look up the CREATE PROCEDURE statement in the Books Online help application that comes with SQL Server.*

Running a Stored Procedure with Arguments from the IDE

Try and run the stored procedure you created in the last exercise and see how the argument affects how it's run. You can try running the stored procedure from either the editor window or the Server Explorer window. The Run dialog box asks you for a value for the strLastName argument. Type **Doe** in the text box and click OK. Now all users with the last name of Doe are returned as the result of the stored procedure.

Using a Stored Procedure with Arguments

The uspGetUsersByLastName stored procedure seems to work, so try and run it from code. Listing 6-3 shows how you would do this.

Listing 6-3. Retrieving Rows from a Stored Procedure with an Input Argument

```
1 public void GetUsersByLastName() {
2     SqlConnection cnnUserMan;
3     SqlCommand cmmUser;
4     SqlDataReader drdUser;
5     SqlParameter prmLastName;
6
7     // Instantiate and open the connection
8     cnnUserMan = new SqlConnection(STR_CONNECTION_STRING);
9     cnnUserMan.Open();
10
11    // Instantiate and initialize command
12    cmmUser = new SqlCommand("uspGetUsersByLastName", cnnUserMan);
13    cmmUser.CommandType = CommandType.StoredProcedure;
14    // Instantiate, initialize and add parameter to command
15    prmLastName = cmmUser.Parameters.Add("@strLastName", SqlDbType.VarChar,
16        50);
```

```

17  // Indicate this is an input parameter
18  prmlastName.Direction = ParameterDirection.Input;
19  // Set the value of the parameter
20  prmlastName.Value = "Doe";
21
22  // Return all users with a last name of Doe
23  drdUser = cmdUser.ExecuteReader();
24 }

```

In Listing 6-3, a **SqlParameter** object specifies the input parameter of the stored procedure. On Lines 15 and 16, I ask the command object to create and associate a parameter with the `@strLastName` argument. The value of this parameter is set to “Doe”, which effectively means that only rows containing a last name of Doe are returned.

As you can see, I have specified that the parameter is an input argument using the **ParameterDirection** enum, although you don’t really have to, because this is the default. Don’t worry too much about parameter and argument; they are essentially the same thing.

Creating a Stored Procedure with Arguments and Return Values

So far I have created stored procedures that return a single value or a result set (rows) and a stored procedure that takes an input argument. In many cases, this is all you want, but sometimes it’s not enough. What if you want a value and a result set returned at the same time? Actually, you may want several values and a result set, but I’m sure you get the idea. In such instances, you can use output arguments.

Actually, you can return as many different values and result sets as you want by including multiple **SELECT** statements after the **AS** clause, but I personally think this approach looks messy. If I return rows and one or more values, I generally use **OUTPUT** arguments for the values. I guess to some extent this is a matter of preference. However, you should be aware that including an output parameter is a faster approach than having it returned in a **DataSet** object, but sometimes you might need the richer functionality of the **DataSet** class, once the values have been returned.

Instead of using the following example code to return a scalar value, two result sets, and another scalar value in that order:

```

...
AS
    SELECT 19
    SELECT * FROM tblUser
    SELECT * FROM tblUserRights
    SELECT 21

```

I would use something like this:

```
...
AS
    SELECT * FROM tblUser
    SELECT * FROM tblUserRights
```

The two return values should then be returned as OUTPUT arguments. But it's your call, my friend, as to which approach you prefer to use. Please note that OUTPUT arguments can also serve as INPUT arguments by default, meaning you can actually supply a value in the OUTPUT argument when calling the stored procedure, and get a different value back. Just like a value passed by reference from one procedure to another.

EXERCISE

Create a new stored procedure and save it with the name **uspGetUsersAndRights**. This stored procedure should return the value 55 for the OUTPUT argument `lngNumRows`, and then all rows in the `tblUser` table and all rows in the `tblUserRights` table.

The stored procedure should look like the one in Figure 6-6.

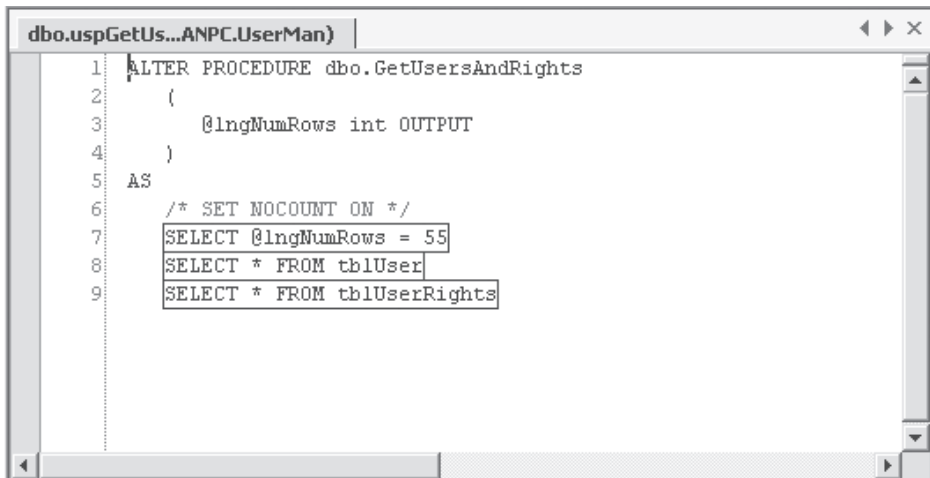


Figure 6-6. The `uspGetUsersAndRights` stored procedure

In the `uspGetUsersAndRights` stored procedure, shown in Figure 6-6, you can see that the `@lngNumRows int` argument is set to the value 55 on Line 7. However, using a default value for the argument you can achieve the same result, by changing Line 3 like this:

```
@lngNumRows int = 55 OUTPUT
```

This means that if for some reason you don't set the value of this parameter when calling the stored procedure or within the stored procedure itself, it'll return 55 as the output value. Default argument values also work for input arguments, and they're specified the same way, using the equal sign followed by the default value, right after the data type.

Running a Stored Procedure with Arguments and Return Values from the IDE

If you've created and saved the stored procedure in the previous exercise, test it by running it. You can try running the stored procedure from either the editor window or the Server Explorer window. The Output window, located just below the editor window, will display the output from the stored procedure, and it should look similar to the output in Figure 6-7.

NOTE *Syntax testing of your stored procedure is done when you save it, and I have a feeling you have already encountered this. If not, just know that's how it is—syntax errors are caught when you try to save your stored procedure.*

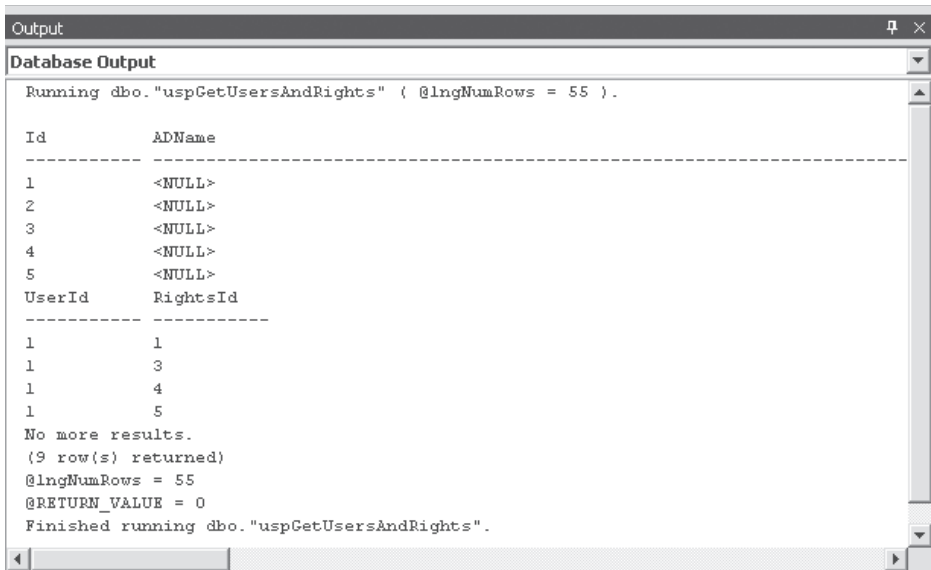


Figure 6-7. The Output window with output from the `uspGetUsersAndRights` stored procedure

Using a Stored Procedure with Arguments and Return Values

Listing 6-4 shows the code to execute the `uspGetUsersAndRights` stored procedure programmatically.

Listing 6-4. Retrieving Rows and Output Values from a Stored Procedure

```
1 public void GetUsersAndRights() {
2     SqlConnection cnnUserMan;
3     SqlCommand cmmUser;
4     SqlDataReader drdUser;
5     SqlParameter prmNumRows;
6
7     // Instantiate and open the connection
8     cnnUserMan = new SqlConnection(STR_CONNECTION_STRING);
9     cnnUserMan.Open();
10
11    // Instantiate and initialize command
12    cmmUser = new SqlCommand("uspGetUsersAndRights", cnnUserMan);
13    cmmUser.CommandType = CommandType.StoredProcedure;
14    // Instantiate, initialize and add parameter to command
15    prmNumRows = cmmUser.Parameters.Add("@lngNumRows", SqlDbType.Int);
16    // Indicate this is an output parameter
```

```

17     prmNumRows.Direction = ParameterDirection.Output;
18     // Get first batch of rows (users)
19     drdUser = cmmUser.ExecuteReader();
20
21     // Display the last name of all user rows
22     while (drdUser.Read()) {
23         MessageBox.Show(drdUser["LastName"].ToString());
24     }
25
26     // Get next batch of rows (user rights)
27     if (drdUser.NextResult()) {
28         // Display the id of all rights
29         while (drdUser.Read()) {
30             MessageBox.Show(drdUser["RightsId"].ToString());
31         }
32     }
33 }

```

In Listing 6-4, two result sets are returned, and therefore I use the **NextResult** method of the `DataReader` class to advance to the second result set on Line 27. Otherwise this stored procedure works pretty much the same as one with input parameters, although the parameter direction is specified as an output on Line 17.

Retrieving a Value Specified with RETURN

In a stored procedure, you can use the `RETURN` statement to return a scalar value. However, this value cannot be retrieved using the **ExecuteScalar** method of the `Command` class, as it would when you use the `SELECT` statement (refer back to Figure 6-2). Of course there is a way of retrieving this value, which I show you after the following exercise.

EXERCISE

Create a new stored procedure and save it with the name **uspGetRETURN_VALUE**. This stored procedure should return the value 55 as the RETURN_VALUE. The stored procedure should look like the one in Figure 6-8.

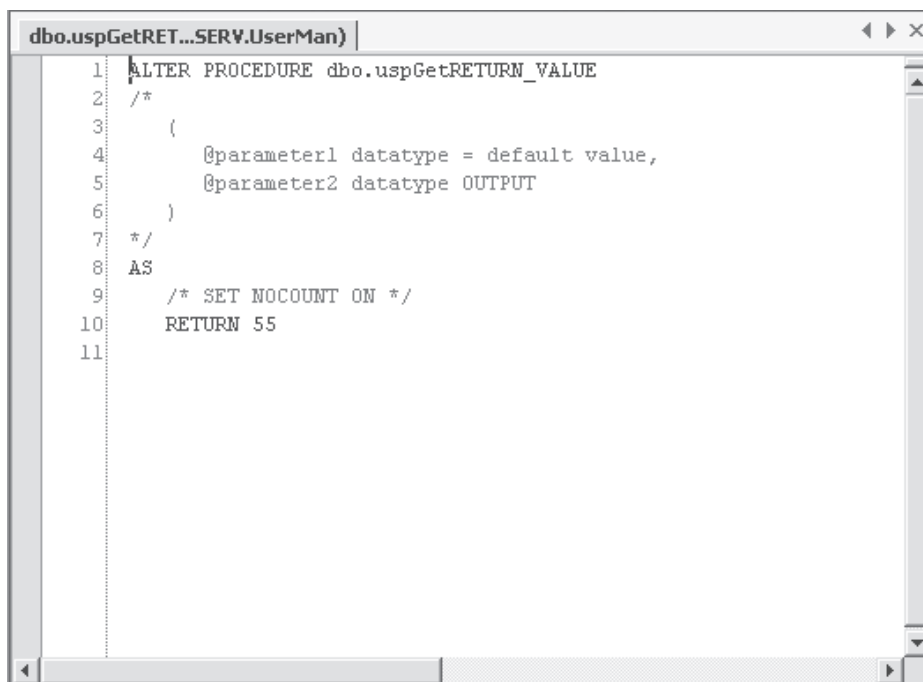


Figure 6-8. The uspGetRETURN_VALUE stored procedure

Listing 6-5 shows you how to retrieve the value from code.

Listing 6-5. Retrieving RETURN_VALUE from a Stored Procedure

```

1 public void GetRETURN_VALUE() {
2     SqlConnection cnnUserMan;
3     SqlCommand cmdUser;
4     SqlParameter prmNumRows;
5     object objResult;
6
7     // Instantiate and open the connection
8     cnnUserMan = new SqlConnection(STR_CONNECTION_STRING);
9     cnnUserMan.Open();
10

```

```

11 // Instantiate and initialize command
12 cmmUser = new SqlCommand("uspGetRETURN_VALUE", cnnUserMan);
13 cmmUser.CommandType = CommandType.StoredProcedure;
14 // Instantiate, initialize and add parameter to command
15 prmNumRows = cmmUser.Parameters.Add("@RETURN_VALUE", SqlDbType.Int);
16 // Indicate this is a return value parameter
17 prmNumRows.Direction = ParameterDirection.ReturnValue;
18 // Get RETURN_VALUE like this, ...
19 objResult = cmmUser.ExecuteScalar();
20 // or like this
21 MessageBox.Show(prmNumRows.Value.ToString());
22 }

```

In Listing 6-5, the **ExecuteScalar** method gets the RETURN_VALUE from a stored procedure. Normally, you would use this method to return the value in the `lngResult` variable, but this variable will contain the default value, 0, in this case. However, because I have specified the **Direction** property of the `prmNumRows` parameter with the **ReturnValue** member of the **ParameterDirection** enum, I can simply look at the **Value** property of the parameter after executing the command.

Changing the Name of a Stored Procedure

If you change the name of your stored procedure in the editor window, the stored procedure is saved with the new name when you save (Ctrl+S). However, if you're not using this method to copy an existing stored procedure, you should be aware that the old stored procedure still exists. So you'll have to delete it if you don't want it.

Viewing Stored Procedure Dependencies

In SQL Server Enterprise Manager, you can see what tables and other objects your stored procedure uses or is dependent on. Open up Enterprise Manager, expand your SQL Server, expand databases and your database, select the Stored Procedures node, right-click the stored procedure you want to see the dependencies for, and select All Task Display Dependencies from the pop-up menu. This brings up the Dependencies dialog box, where you can see what database objects your stored procedure depends on and vice versa. This is also called the *ownership chain*.

Running Oracle Stored Procedures

Oracle stored procedures and stored functions are different from those of SQL Server. That's why I've chosen to explain the Oracle stored procedures in a separate section. When discussing what you can do with stored procedures/functions in Oracle compared to stored procedures in SQL Server, it's pretty much the same, but the implementation is quite different.

SQL Server stored procedures can return a value, just like a function in C#, whereas in Oracle you have stored procedures and stored functions. This means that if you want a return value that isn't a parameter, you must use a stored function. In this chapter, you won't see how to create stored procedures and stored functions in Oracle, but you can use the Oracle Database Project located in the example code, which you can download from the Apress Web site (<http://www.apress.com>) or the UserMan site (<http://www.userman.dk>), to create the tables, stored procedures, views, and triggers used by the Oracle example code. Please consult your Oracle documentation if you need more information on how to implement stored procedures and stored functions in Oracle.

When you use ADO.NET and ADO for that matter, you can't use the **ExecuteScalar** method of the `DataReader` class to retrieve a return value, as shown in Listings 6-1 and 6-5 and discussed in the "Retrieving a Value Specified with RETURN" section. This is also true if you execute a stored function. You need to return any return values in output parameters, just as I've demonstrated in Listing 6-4. If you only need to return a value, as in Listing 6-1, which is what the Oracle stored function in Listing 6-6 does, you can do as is shown in Listing 6-7, which is really more or less the same code as in Listing 6-1.

Listing 6-6. A Simple Oracle Stored Function

```
1 CREATE OR REPLACE FUNCTION SIMPLESTOREDFUNCTION
2 RETURN NUMBER
3 AS
4     lngNumRows NUMBER;
5 BEGIN
6     SELECT COUNT(*) INTO lngNumRows FROM TBLUSER;
7     RETURN lngNumRows;
8 END SIMPLESTOREDFUNCTION;
```

In Listing 6-6, you can see an Oracle stored function that returns the number of rows in the `tblUser` table. You can see in Listing 6-7 how you can access this stored function and retrieve the return value.

Listing 6-7. Running a Simple Oracle Stored Function

```

1 public void ExecuteSimpleOracleSF() {
2     OleDbConnection cnnUserMan;
3     OleDbCommand cmmUser;
4     OleDbParameter prmNumRows;
5     object objReturnValue;
6
7     // Instantiate and open the connection
8     cnnUserMan = new OleDbConnection(STR_CONNECTION_STRING);
9     cnnUserMan.Open();
10
11    // Instantiate and initialize command
12    cmmUser = new OleDbCommand("SimpleStoredFunction", cnnUserMan);
13    cmmUser.CommandType = CommandType.StoredProcedure;
14    // Instantiate output parameter and add to parameter
15    // collection of command object
16    prmNumRows = cmmUser.CreateParameter();
17    prmNumRows.Direction = ParameterDirection.ReturnValue;
18    prmNumRows.DbType = DbType.Int64;
19    prmNumRows.Precision = 38;
20    prmNumRows.Size = 38;
21    cmmUser.Parameters.Add(prmNumRows);
22
23    // Retrieve and display value
24    objReturnValue = cmmUser.ExecuteScalar();
25    MessageBox.Show(cmmUser.Parameters[0].Value.ToString());
26 }

```

In Listing 6-7, I've actually used the **ExecuteScalar** method of the `DataReader` class on Line 24, but if you look carefully, you'll see that I don't use the value returned from the function call (`objReturnValue`) as in Listing 6-1. However, I do retrieve the return value in the `prmNumRows` parameter, which is instantiated, initialized, and set up as a return value on Lines 12 through 21, and display it after executing the command on Line 25. I use **ExecuteScalar** method, because it has the least overhead of any of the **Execute** methods of the `DataReader` class. So even if you don't use the return value from the **ExecuteScalar** method, which is always **null** when calling an Oracle stored function or stored procedure, you can still get the return value from the stored function. The trick is add a parameter to the command object and make sure you set the **Direction** property of the parameter object to the **ReturnValue** member of the **ParameterDirection** enum, as is shown on Line 13.

If you want to use an Oracle stored procedure instead of a stored function, like the one shown in Listing 6-8, to retrieve one or more simple data types using output parameters, you can use the example code shown in Listing 6-9.

Listing 6-8. A Simple Oracle Stored Procedure

```
1 CREATE OR REPLACE PROCEDURE SIMPLESTOREDPROCEDURE
2 (lngNumRows OUT NUMBER)
3 AS
4 BEGIN
5     SELECT COUNT(*) INTO lngNumRows FROM TBLUSER;
6 END SIMPLESTOREDPROCEDURE;
```

The Oracle stored procedure in Listing 6-8 accepts one output parameter (lngNumRows) and sets this parameter to the number of rows in the tblUser table when executed. You can see how you can call this stored procedure from code, in Listing 6-9.

Listing 6-9. Running a Simple Oracle Stored Procedure

```
1 public void ExecuteSimpleOracleSP() {
2     OleDbConnection cnnUserMan;
3     OleDbCommand cmdUser;
4     OleDbParameter prmNumRows;
5     object objReturnValue;
6
7     // Instantiate and open the connection
8     cnnUserMan = new OleDbConnection(STR_CONNECTION_STRING);
9     cnnUserMan.Open();
10
11    // Instantiate and initialize command
12    cmdUser = new OleDbCommand("SimpleStoredProcedure", cnnUserMan);
13    cmdUser.CommandType = CommandType.StoredProcedure;
14    // Instantiate output parameter and add to parameter
15    // collection of command object
16    prmNumRows = cmdUser.CreateParameter();
17    prmNumRows.Direction = ParameterDirection.Output;
18    prmNumRows.DbType = DbType.Int64;
19    prmNumRows.Precision = 38;
20    prmNumRows.Size = 38;
21    cmdUser.Parameters.Add(prmNumRows);
22
23    // Retrieve and display value
24    objReturnValue = cmdUser.ExecuteScalar();
25    MessageBox.Show(cmdUser.Parameters[0].Value.ToString());
26 }
```

In Listing 6-9, I again use the **ExecuteScalar** method of the `DataReader` class on Line 24 for retrieving a value from a stored procedure. The example code on Listing 6-9 really isn't all that different from Listing 6-7, but it does show you how to call a stored procedure instead of a stored function.

The Oracle stored procedures and stored functions, and the example code to execute them shown so far, only deal with simple return values. If you need to return result sets, such as in Listings 6-2, 6-3, and 6-4, you need to use cursors in the stored procedures.⁴ Listing 6-10 shows a stored procedure that returns a result set using cursors.

Listing 6-10. Oracle Stored Procedure Returning Result Set

```

1 CREATE OR REPLACE PACKAGE PKGTBLUSER
2 AS
3     TYPE CUR_TBLUSER IS REF CURSOR RETURN TBLUSER%ROWTYPE;
4 END PKGTBLUSER;
5
6 CREATE OR REPLACE PROCEDURE USPGETUSERSBYLASTNAME
7     (ROWS OUT PKGTBLUSER.CUR_TBLUSER, strLastName IN VARCHAR2)
8 IS
9 BEGIN
10 OPEN ROWS FOR SELECT * FROM TBLUSER
11 WHERE LASTNAME = strLastName;
12 END USPGETUSERSBYLASTNAME;
```

In Listing 6-10, you can see how I first create a package definition (Lines 1 through 4) in my Oracle database, and in this package I define the `CUR_TBLUSER` cursor type, which is of data type **REF CURSOR**,⁵ that returns rows from the `tblUser` table. The package definition only holds the type declaration, which is used in the stored procedure. Please note that the notion of Oracle package definitions and package bodies are beyond the scope of this book, although you can certainly use a package body instead of the stored procedure shown on Lines 6 through 12. Please see your Oracle documentation for more information on packages.

You need to declare the cursor type in a package, because you're using it as the data type for one of the parameters in the `USPGETUSERSBYLASTNAME` stored procedure. You can't declare a data type in the parameters section of a stored

4. You can also use cursors with a stored function, but because I won't be using the function return value from this point on, I'll concentrate on using stored procedures.

5. This is short for REFERENCE CURSOR, and basically it's used as a pointer to the original data. Please see your Oracle documentation for more information.

procedure, which is why you need it declared elsewhere. If you look at the parameters declaration on Line 7, you can see that I need to use the full path to the data type, PKGTBLUSER.CUR_TBLUSER. Lines 10 and 11 of Listing 6-10 is where the rows that match the passed last name criterion, are retrieved with the CUR_TBLUSER cursor and saved in the ROWS OUT parameter. Listing 6-11 shows you how to retrieve the result set from the stored procedure.

Listing 6-11. Retrieving Result Set from Oracle Stored Procedure

```

1 public void OracleGetUsersByLastName() {
2     OleDbConnection cnnUserMan;
3     OleDbCommand cmmUser;
4     OleDbParameter prmLastName;
5     OleDbDataReader drdUser;
6
7     // Instantiate and open the connection
8     cnnUserMan = new OleDbConnection(STR_CONNECTION_STRING);
9     cnnUserMan.Open();
10
11    // Instantiate and initialize command
12    cmmUser = new OleDbCommand("USPGETUSERSBYLASTNAME", cnnUserMan);
13    cmmUser.CommandType = CommandType.StoredProcedure;
14    // Instantiate, initialize and add parameter to command
15    prmLastName = cmmUser.Parameters.Add("strLastName", OleDbType.VarChar,
16        50);
17    // Indicate this is an input parameter
18    prmLastName.Direction = ParameterDirection.Input;
19    // Set the type and value of the parameter
20    prmLastName.Value = "Doe";
21
22    // Retrieve rows
23    drdUser = cmmUser.ExecuteReader();
24    // Loop through the returned rows
25    while (drdUser.Read()) {
26        // Display the last name of all user rows
27        MessageBox.Show(drdUser["LastName"].ToString());
28    }
29 }
```

In Listing 6-11, you can see how I set up the command object on Lines 12 through 13, and then prepare the prmLastName input parameter with the value of “Doe”. I then call the **ExecuteReader** method of the Command class, which returns the DataReader with all users with a last name of Doe. If you compare the stored procedure in Listing 6-10 and the example code in Listing 6-11, you’ll see

that there's a mismatch of the number of parameters. The stored procedure has two parameters, the last name input parameter and the result set output parameter. However, I only set up one parameter in Listing 6-11 and that's the last name input parameter. The command object takes care of returning the result set as the return value of the function call (**ExecuteReader**) instead of as an output parameter. It almost works the same as with the SQL Server example code in Listing 6-3.

NOTE *It doesn't matter where you place the `ROWS OUT` parameter in the stored procedure parameter declaration—that is, whether you place it first as is done in Listing 6-10, or last like this:*

```
(strLastName IN VARCHAR2, ROWS OUT PKGTBLUSER.CUR_TBLUSER)
```

There are other ways of calling a stored procedure in your Oracle database, such as using the ODBC `{call storedprocedurename}` syntax, but I've chosen to show you the way that looks and feels as close to the one used for calling SQL Server procedures.

Using Views

A *view* is, as the word suggests, a display of data in your database. Perhaps it helps to think of a view as a virtual table. It can be a subset of a table or an entire table, or it can be a subset of several joined tables. Basically, a view can represent just about any subset of data in your database, and you can include other views in a view. Including a view in another view is called *nesting*, and it can be a valuable way of grouping display data. However, nesting too deeply can also result in performance problems and can certainly make it a real challenge to track down errors. There isn't really any magic to a view or any big secrets that I can let you in on; it's simply just a great tool for manipulating your data. In the rest of this section, I am going to look at why, when, where, and how you should use a view.

NOTE *The example code shown in this section is SQL Server only, but if you take a look at the accompanying example code, you'll see that it works exactly the same with Microsoft Access queries. Only the SQL Server .NET Data Provider has been changed to the OLE DB .NET Data Provider. The same goes for Oracle views. See the example code, which is almost identical to the Microsoft Access code. Views aren't supported in MySQL 3.23.45.*

View Restrictions

A view is almost identical to a row-returning query, with just a few exceptions. Some of the restrictions are detailed here:

- COMPUTE and COMPUTE BY clauses cannot be included in your view.
- ORDER BY clauses aren't allowed in a view, unless you specify the TOP clause as part of the SELECT statement. However, you can index a view with SQL Server 2000.
- The INTO keyword cannot be used to create a new table.
- Temporary tables cannot be referenced.

There are other restrictions, so please check with your SQL Server documentation and/or Help Files.

Why Use a View?

Like stored procedures, views are used for server-side processing of your data, but whereas stored procedures mainly are used for security and performance reasons, views are generally used to secure access to your data and to hide complexity of queries that contain many joins. You may want to use a view for a variety of reasons:

- *Security:* You don't want your users to access the tables directly, and with the help of a view you can restrict users to seeing only the parts of a table they are allowed to see. You can restrict access to specific columns and/or rows and thus make it easy for your users to use for their own queries.
- *Encryption:* You can encrypt a view so that no one can see where the underlying data comes from. Mind you, this is an irreversible action, meaning that the textual SQL statements that form the view can't be retrieved again!
- *Aggregated data:* Views are often used on large scale systems to provide aggregated data.

There are other reasons for creating a view, but the mentioned reasons are certainly two of the most common.

Creating a View

It's easy to create a view. If you are used to working with the SQL Server's **Server Manager**, you should check out what the **Server Explorer** has to offer you. Here's how you create a view using the UserMan database as an example:

1. Open up the Server Explorer window.
2. Expand the UserMan database on your database server.
3. Right-click the Views node and select New View.

This brings up the View Designer, which in fact is the same as the Query Designer.

NOTE *The Query Designer is described in detail in Chapter 4. Although the View Designer and Query Designer have the same look and feel, you cannot create views that don't adhere to the view restrictions mentioned in the section, "View Restrictions."*

The Add Table dialog box is also shown when the View Designer is displayed. In this dialog box, simply select the tables you want to retrieve data from and click Add. Click Close when all the required tables have been added.

As you start selecting in the Diagram pane the columns that should be output when the view is run, the SQL pane and the Grid pane change accordingly. When you are done selecting the columns to output, you should save the view using Ctrl+S.

EXERCISE

- 1) Create a new view. This view should contain the following tables: tblUser, tblRights, and tblUserRights. The following fields should be output: tblUser.LoginName, tblUser.FirstName, tblUser.LastName, and tblRights.Name.
- 2) Save the view under the name **viewUserInfo**. The new view should look like the one in Figure 6-9.

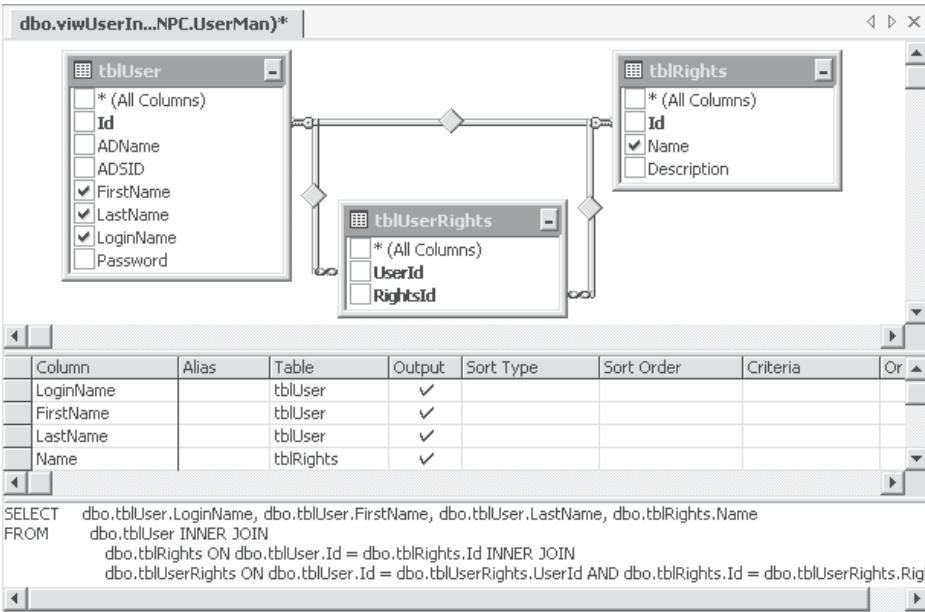


Figure 6-9. The viewUserInfo view

Running a View from the IDE

“Running a view” is perhaps not the most appropriate phrase when you think about it. On the other hand, the view does have to retrieve the data from all the tables referenced in the view, so I guess this phrase will have to do.

Anyway, you can run a view from the View Designer by right-clicking a blank area of the View Designer and selecting Run from the pop-up menu. The data retrieved by the view is then displayed in the Results pane of the View Designer.

EXERCISE

- 1) Run the `vwUserInfo` view from the View Designer. The Results pane now displays rows like the ones in Figure 6-10.

	LoginName	FirstName	LastName	Name
▶	UserMan	John	Doe	AddUser
✱				

Figure 6-10. The results of running the `vwUserInfo` view

- 2) Notice that the Name field of the `tblRights` table seems a bit confusing, because it doesn't really show what Name means. So in the Grid pane, you should add the text **RightsName** to the Alias column in the Name row.
- 3) Run the view again and notice how the new column name appears in the Results pane.
- 4) Save the view with `Ctrl+S`.

Using a View from Code

Actually, it's very easy to use a view in code, because a view is referenced like any standard table in your database, which means that you can retrieve data using a command object or a data adapter that fills a data set, and so on.

NOTE Please see Chapter 3B for specific information on how to manipulate data in a table.

Retrieving Read-Only Data from a View in Code

The simplest use of a view is for display purposes, like when you just need to display some information from one or more related tables. Because in the example code I don't have to worry about updates, I don't have to set up anything particular. Listing 6-12 demonstrates how to return all rows from a view and populate a data reader.

Listing 6-12. Retrieving Rows in a View

```
1 public void RetrieveRowsFromView() {
2     SqlConnection cnnUserMan;
3     SqlCommand cmmUser;
4     SqlDataReader drdUser;
5
6     // Instantiate and open the connection
7     cnnUserMan = new SqlConnection(STR_CONNECTION_STRING);
8     cnnUserMan.Open();
9
10    // Instantiate and initialize command
11    cmmUser = new SqlCommand("SELECT * FROM viwUserInfo", cnnUserMan);
12    // Get rows
13    drdUser = cmmUser.ExecuteReader();
14 }
```

Listing 6-12 is just like any other row-returning query, except that a view is queried instead of a table.

Manipulating Data in a View from Code

Listing 6-12 shows you how to retrieve data from a view into a data reader, and this means the data cannot be updated, because the data reader doesn't allow updates. However, it's possible to update data in a view. The problem with this is that various versions of SQL Server support different levels of update support for views. If you only have one table in a view, this isn't a problem at all; however, if you have multiple tables in a view, only SQL Server 2000 supports updating rows in more than one of the source tables. Besides the mentioned problems, you'll certainly run into even bigger problems if you need to migrate to a different RDBMS. Generally, I would discourage updating data in views.

EXERCISE

- 1) Create a new view. This view should contain the tblUser table. The following fields should be output: Id, FirstName, LastName, and LoginName.
- 2) Save the view under the name **viwUser**. The new view should look like the one in Figure 6-11.

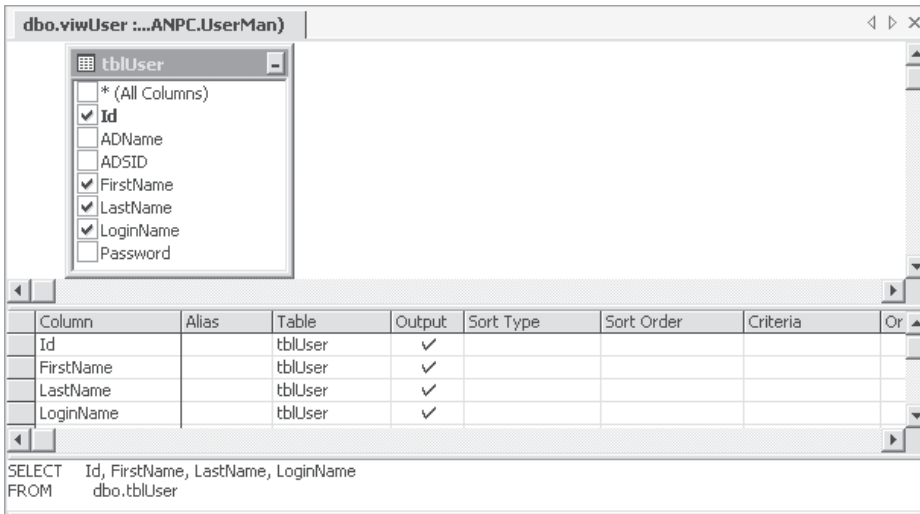


Figure 6-11. The viwUser view

This view, which can be located on SQL Server 7.0 as well as SQL Server 2000, can be manipulated using the code in Listing 6-13.

Listing 6-13. Manipulating Data in a View Based on a Single Table

```
1 public void ManipulatingDataInViewBasedOnSingleTable() {
2     const string STR_SQL_USER_SELECT = "SELECT * FROM viwUser";
3     const string STR_SQL_USER_DELETE = "DELETE FROM viwUser WHERE Id=@Id";
4     const string STR_SQL_USER_INSERT = "INSERT INTO viwUser(FirstName, " +
5         "LastName, LoginName, Logged, Description) VALUES(@FirstName, " +
6         "@LastName, @LoginName)";
7     const string STR_SQL_USER_UPDATE = "UPDATE viwUser SET FirstName=" +
8         "@FirstName, LastName=@LastName, LoginName=@LoginName WHERE Id=@Id";
9
10    SqlConnection cnnUserMan;
11    SqlCommand cmmUser;
```

```
12     SqlDataAdapter dadUser;
13     DataSet dstUser;
14
15     SqlCommand cmmUserSelect;
16     SqlCommand cmmUserDelete;
17     SqlCommand cmmUserInsert;
18     SqlCommand cmmUserUpdate;
19
20     SqlParameter prmSQLDelete, prmSQLUpdate;
21
22     // Instantiate and open the connection
23     cnnUserMan = new SqlConnection(STR_CONNECTION_STRING);
24     cnnUserMan.Open();
25
26     // Instantiate and initialize command
27     cmmUser = new SqlCommand("SELECT * FROM viwUser", cnnUserMan);
28     // Instantiate the commands
29     cmmUserSelect = new SqlCommand(STR_SQL_USER_SELECT, cnnUserMan);
30     cmmUserDelete = new SqlCommand(STR_SQL_USER_DELETE, cnnUserMan);
31     cmmUserInsert = new SqlCommand(STR_SQL_USER_INSERT, cnnUserMan);
32     cmmUserUpdate = new SqlCommand(STR_SQL_USER_UPDATE, cnnUserMan);
33     // Instantiate command and data set
34     cmmUser = new SqlCommand(STR_SQL_USER_SELECT, cnnUserMan);
35     dstUser = new DataSet();
36
37     dadUser = new SqlDataAdapter();
38     dadUser.SelectCommand = cmmUserSelect;
39     dadUser.InsertCommand = cmmUserInsert;
40     dadUser.DeleteCommand = cmmUserDelete;
41     dadUser.UpdateCommand = cmmUserUpdate;
42
43     // Add parameters
44     prmSQLDelete = dadUser.DeleteCommand.Parameters.Add("@Id", SqlDbType.Int,
45     0, "Id");
46     prmSQLDelete.Direction = ParameterDirection.Input;
47     prmSQLDelete.SourceVersion = DataRowVersion.Original;
48
49     cmmUserUpdate.Parameters.Add("@FirstName", SqlDbType.VarChar, 50,
50     "FirstName");
51     cmmUserUpdate.Parameters.Add("@LastName", SqlDbType.VarChar, 50,
52     "LastName");
53     cmmUserUpdate.Parameters.Add("@LoginName", SqlDbType.VarChar, 50,
54     "LoginName");
```

```

55     prmSQLUpdate = dadUser.UpdateCommand.Parameters.Add("@Id", SqlDbType.Int,
56         0, "Id");
57     prmSQLUpdate.Direction = ParameterDirection.Input;
58     prmSQLUpdate.SourceVersion = DataRowVersion.Original;
59
60     cmmUserInsert.Parameters.Add("@FirstName", SqlDbType.VarChar, 50,
61         "FirstName");
62     cmmUserInsert.Parameters.Add("@LastName", SqlDbType.VarChar, 50,
63         "LastName");
64     cmmUserInsert.Parameters.Add("@LoginName", SqlDbType.VarChar, 50,
65         "LoginName");
66
67     // Populate the data set from the view
68     dadUser.Fill(dstUser, "viwUser");
69
70     // Change the last name of user in the second row
71     dstUser.Tables["viwUser"].Rows[1]["LastName"] = "Thomsen";
72     dstUser.Tables["viwUser"].Rows[1]["FirstName"] = "Carsten";
73     // Propagate changes back to the data source
74     dadUser.Update(dstUser, "viwUser");
75 }

```

In Listing 6-13, a data adapter and a data set were set up to retrieve and hold data from the `viwUser` view. The `LastName` column of row 2 is then updated as well as the data source with the changes in the data set. This simple demonstration was designed to show you how to work with views based on a single table.

Using Triggers

A *trigger* is actually a stored procedure that automatically invokes (triggers) when a certain change is applied to your data. Triggers are the final server-side processing functionality that I'll discuss in this chapter. Until SQL Server 2000 was released, triggers were a vital part of enforcing referential integrity, but with the release of SQL Server 2000, you now have that capability built in. In the rest of this section, I'll show you what a trigger is and when and how you can use it, but there is little C# programming involved with using triggers, because they operate entirely internally, only passing status or error indicators back to the client.

Triggers respond to data modifications using `INSERT`, `UPDATE`, and `DELETE` operations. Basically, you can say that a trigger helps you write less code; you can incorporate business rules as triggers and thus prevent the inclusion of data that is invalid because it violates your business rules.

SQL Server implements AFTER triggers, meaning that the trigger is invoked after the modification has occurred. However, this doesn't mean that a change can't be rolled back, because the trigger has direct access to the modified row and as such can roll back any modification. When SQL Server 2000 was released you also got support for the notion of BEFORE triggers, which you might know from the Oracle RDBMS. In SQL Server 2000, they are called INSTEAD OF triggers.

NOTE *The example code shown in this section is SQL Server only, but if you take a look at the accompanying example code, you'll see that Oracle after triggers work almost the same as SQL Server triggers, although the syntax is different. Only the SQL Server .NET Data Provider has been changed to the OLE DB .NET Data Provider. Triggers aren't supported in Microsoft Access or MySQL 3.23.45.*

Why Use a Trigger?

Triggers are automatic, so you don't have to apply the business logic in your code. Here's a perfect situation for a business rule: you need to check if a member of an organization has paid his or her annual fee and therefore is allowed to order material from the organization's library. An INSERT trigger could perform the lookup in the members table when a member tries to order material and check if the member has paid the annual fee. This is exactly what makes a trigger more useful than a constraint in some situations, because a trigger can access columns in other tables, unlike a constraint, which can only access columns in the current table or row. If your code is to handle your business rule, this would mean that you need to look up the member's information in the members table before you can insert the order in the orders table. With the trigger, this lookup is done automatically, and an exception is thrown if you try to insert an order for library material if the member hasn't paid his or her annual fee. Furthermore, you don't have to rely on another front-end code developer to know what the business rules are.

In short, use a trigger for keeping all your data valid or to comply with your business rules. Think of triggers as an extra validation tool, while at the same time making sure you have set up referential integrity.

NOTE *With SQL Server 2000, you shouldn't use triggers for referential integrity (see Chapter 2), because you can set that up with the Database Designer. See Chapter 4 for information on the Database Designer.*

Creating a Trigger

It's quite easy to create a trigger. This can be done using the Server Manager that comes with SQL Server, but I'll use the Server Explorer. Here's how you create a trigger for the example UserMan database:

1. Open up the Server Explorer window.
2. Expand the UserMan database on your database server.
3. Expand the Tables node.
4. Right-click the table for which you want to create a trigger and select New Trigger from the pop-up menu.

This brings up the trigger text editor, which is more or less the same editor you use for your C# code (see Figure 6-12).

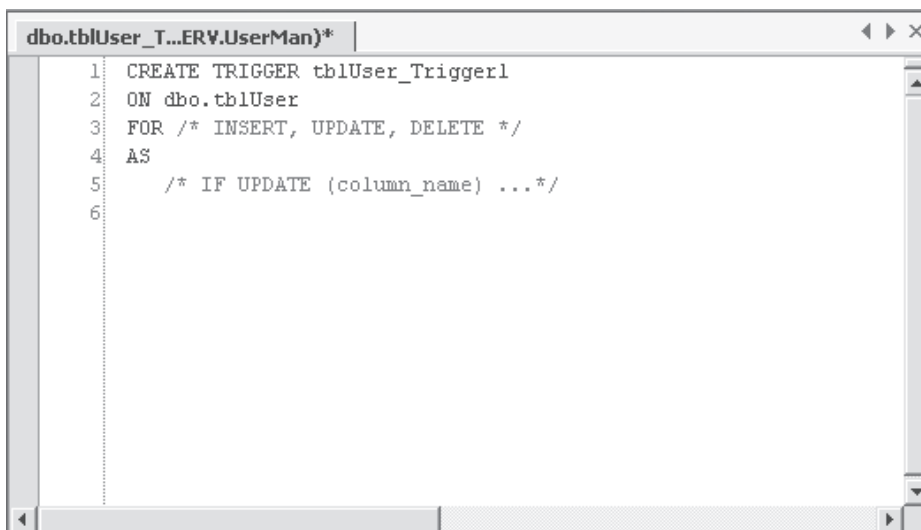


Figure 6-12. Trigger editor with default template

In the trigger editor, you can see that the template automatically names a new trigger Trigger1 prefixed with the name of the table. Actually, if another trigger with this name already exists, the new trigger is named Trigger2, and so on.

Once you are done editing your trigger, you need to save it by pressing Ctrl+S. As soon as you have saved it, the very first line of the stored procedure changes. The SQL statement CREATE TRIGGER is changed so that the first line reads as follows:

```
ALTER TRIGGER dbo. . . .
```

NOTE *The trigger editor performs syntax checking when you save your trigger, meaning if the syntax of your trigger is invalid, you aren't allowed to save it to your database.*

EXERCISE

- 1) Create a new trigger for the tblUser table and save it with the name **tblUser_Update**.
- 2) This is an update trigger, so you need to change the text on Line 3 to **FOR UPDATE**.
- 3) Replace the text on Line 5 and down with the following:

```
DECLARE @strFirstName varchar(50)  
/* Get the value for the FirstName column */  
SELECT @strFirstName = (SELECT FirstName FROM inserted)  
/* Check if we're updating the LastName column.  
If so, make sure FirstName is not NULL */  
IF UPDATE (LastName) AND @strFirstName IS NULL  
BEGIN  
/* Roll back update and raise exception */  
ROLLBACK TRANSACTION  
RAISERROR ('You must fill in both LastName and FirstName', 11, 1)  
END
```

Now the stored procedure should look like the one in Figure 6-13.

```

1 ALTER TRIGGER tblUser_Update
2 ON dbo.tblUser
3 FOR UPDATE
4 AS
5     DECLARE @strFirstName varchar(50)
6     /* Get the value for the FirstName column */
7     SELECT @strFirstName = (SELECT FirstName FROM inserted)
8     /* Check if we're updating the LastName column.
9     If so, make sure FirstName is not NULL */
10    IF UPDATE (LastName) AND @strFirstName IS NULL
11    BEGIN
12        /* Roll back update and raise exception */
13        ROLLBACK TRANSACTION
14        RAISERROR ('You must supply both LastName and FirstName', 11, 1)
15    END
16

```

Figure 6-13. The tblUser_Update trigger

4) Don't forget to save the changes using Ctrl+S.

When a trigger has been saved to the database, you can locate it under the table to which it belongs in the **Server Explorer**.

The tblUser_Update trigger is invoked when updating a row in the user table. The trigger first tests to see if the LastName column is updated. If it is, then the trigger checks to see if the FirstName column is empty, because if it is the update is rolled back and an exception is raised. Please note that this trigger is designed to work with only one updated or inserted row at a time. If more rows are inserted at the same time, the trigger will have to be redesigned to accommodate this. However, this trigger only serves as a demonstration. The functionality of this trigger can easily be implemented using constraints, because the check I perform is done in the same table. If I had looked up a value in a different table, then the trigger would be your only choice.

Please see your SQL Server documentation if you need more information on how to create triggers. Listing 6-14 shows you how to execute the new trigger, demonstrating how to raise an exception you can catch in code.

Listing 6-14. Invoking Trigger and Catching Exception Raised

```

1 public void TestUpdateTrigger() {
2     const string STR_SQL_USER_SELECT = "SELECT * FROM tblUser";
3     const string STR_SQL_USER_DELETE = "DELETE FROM tblUser WHERE Id=@Id";
4     const string STR_SQL_USER_INSERT = "INSERT INTO tblUser(FirstName, " +
5         "LastName, LoginName) VALUES(@FirstName, @LastName, @LoginName)";
6     const string STR_SQL_USER_UPDATE = "UPDATE tblUser SET " +
7         "FirstName=@FirstName, LastName=@LastName, LoginName=@LoginName WHERE " +
8         "Id=@Id";
9
10    SqlConnection cnnUserMan;
11    SqlCommand cmmUser;
12    SqlDataAdapter dadUser;
13    DataSet dstUser;
14
15    SqlCommand cmmUserSelect;
16    SqlCommand cmmUserDelete;
17    SqlCommand cmmUserInsert;
18    SqlCommand cmmUserUpdate;
19
20    SqlParameter prmSQLDelete, prmSQLUpdate, prmSQLInsert;
21
22    // Instantiate and open the connection
23    cnnUserMan = new SqlConnection(STR_CONNECTION_STRING);
24    cnnUserMan.Open();
25
26    // Instantiate and initialize command
27    cmmUser = new SqlCommand("SELECT * FROM tblUser", cnnUserMan);
28    // Instantiate the commands
29    cmmUserSelect = new SqlCommand(STR_SQL_USER_SELECT, cnnUserMan);
30    cmmUserDelete = new SqlCommand(STR_SQL_USER_DELETE, cnnUserMan);
31    cmmUserInsert = new SqlCommand(STR_SQL_USER_INSERT, cnnUserMan);
32    cmmUserUpdate = new SqlCommand(STR_SQL_USER_UPDATE, cnnUserMan);
33    // Instantiate command and data set
34    cmmUser = new SqlCommand(STR_SQL_USER_SELECT, cnnUserMan);
35    dstUser = new DataSet();
36
37    dadUser = new SqlDataAdapter();
38    dadUser.SelectCommand = cmmUserSelect;
39    dadUser.InsertCommand = cmmUserInsert;
40    dadUser.DeleteCommand = cmmUserDelete;
41    dadUser.UpdateCommand = cmmUserUpdate;
42

```

```

43 // Add parameters
44 prmSQLDelete = dadUser.DeleteCommand.Parameters.Add("@Id", SqlDbType.Int,
45     0, "Id");
46 prmSQLDelete.Direction = ParameterDirection.Input;
47 prmSQLDelete.SourceVersion = DataRowVersion.Original;
48
49 cmmUserUpdate.Parameters.Add("@FirstName", SqlDbType.VarChar, 50,
50     "FirstName");
51 cmmUserUpdate.Parameters.Add("@LastName", SqlDbType.VarChar, 50,
52     "LastName");
53 cmmUserUpdate.Parameters.Add("@LoginName", SqlDbType.VarChar, 50,
54     "LoginName");
55 prmSQLUpdate = dadUser.UpdateCommand.Parameters.Add("@Id", SqlDbType.Int,
56     0, "Id");
57 prmSQLUpdate.Direction = ParameterDirection.Input;
58 prmSQLUpdate.SourceVersion = DataRowVersion.Original;
59
60 cmmUserInsert.Parameters.Add("@FirstName", SqlDbType.VarChar, 50,
61     "FirstName");
62 cmmUserInsert.Parameters.Add("@LastName", SqlDbType.VarChar, 50,
63     "LastName");
64 cmmUserInsert.Parameters.Add("@LoginName", SqlDbType.VarChar, 50,
65     "LoginName");
66
67 // Populate the data set from the view
68 dadUser.Fill(dstUser, "tblUser");
69
70 // Change the name of user in the second row
71 dstUser.Tables["tblUser"].Rows[1]["LastName"] = "Thomsen";
72 dstUser.Tables["tblUser"].Rows[1]["FirstName"] = null;
73
74 try {
75     // Propagate changes back to the data source
76     dadUser.Update(dstUser, "tblUser");
77 }
78 catch (Exception objE) {
79     MessageBox.Show(objE.Message);
80 }
81 }

```

In Listing 6-14, the second row is updated, and the LastName column is set to “Thomsen” and the FirstName to a **null** value. This will invoke the update trigger that throws an exception, which is caught in code and displays the error message.

This should give you a taste for using triggers, and they really aren't that hard to work with. Just make sure you have a well-designed database that doesn't use triggers for purposes that can easily be achieved by other means such as referential integrity.

Viewing Trigger Source

In SQL Server Enterprise Manager, you can see the source for your triggers. Open up Enterprise Manager, expand your SQL Server, and expand databases and your database. Next, select the Tables node, right-click the table you have created the trigger for, and select All Task Manage Triggers from the pop-up menu. This brings up the Triggers Properties dialog box, where you can see and edit the triggers for the selected table.

In the Server Explorer in the VS .NET IDE, you can also view the trigger source by expanding your database, expanding the Tables node, expanding the table with the trigger, and double-clicking the trigger.

Summary

In this chapter, I discussed how to create various server-side objects for server-side processing of your data. I demonstrated stored procedures, views, and triggers, and showed you how to create, run, and execute a stored procedure from code; how to create, run, and use a view from code, including updating the view; and finally how to create triggers.

I went into enough details about stored procedures, views, and triggers as to what a C# programmer needs to know, but if you are also responsible for coding the SQL Server database and you need more information and example code, I can certainly recommend you read this book:

- *Code Centric: T-SQL Programming with Stored Procedures and Triggers*, by Garth Wells. Published by Apress, February 2001. ISBN: 1893115836.

The next chapter is about hierarchical databases. I'll discuss how you use the LDAP protocol to access a network directory database like the Active Directory and you'll see how to access information stored on Exchange Server 2000.